How to write a financial contract

S.L. Peyton Jones and J-M. Eber

This chapter is based closely on "Composing contracts: an adventure in financial engineering", Proceedings International Conference on Functional Programming, Montreal, 2000, with permission from ACM, New York.

1 Introduction

The finance and insurance industry manipulates increasingly complex contracts. Here is an example: the contract gives the holder the right to choose on 30 June 2000 between

D_1 Both of:

- D_{11} Receive £100 on 29 Jan 2001.
- D_{12} Pay £105 on 1 Feb 2002.

D_2 An option exercisable on 15 Dec 2000 to choose one of:

- D_{21} Both of:
  - D_{211} Receive £100 on 29 Jan 2001.
  - D_{212} Pay £106 on 1 Feb 2002.
- D_{22} Both of:
  - D_{221} Receive £100 on 29 Jan 2001.
  - D_{222} Pay £112 on 1 Feb 2003.

The details of this contract — call it C — are not important, but it is a simplified but realistic example of the sort of contract that is traded in financial derivative markets. What is important is that complex contracts, such as C, are formed by combining together simpler contracts, such as D_1, which in turn are formed from simpler contracts still, such as D_{11}, D_{12}.

At this point, any red-blooded functional programmer should start to foam at the mouth, yelling "build a combinator library". And indeed, that turns out to be not only possible, but tremendously beneficial.

The finance industry has an enormous vocabulary of jargon for typical combinations of financial contracts (swaps, futures, caps, floors, swaptions, spreads,

straddles, captions, European options, American options, ...the list goes on). Treating each of these individually is like having a large catalogue of prefabricated components. The trouble is that someone will soon want a contract that is not in the catalogue.

If, instead, we could define each of these contracts using a fixed, precisely-specified set of combinators, we would be in a much better position than having a fixed catalogue. For a start, it becomes much easier to *describe* new, unforeseen, contracts. Beyond that, we can systematically *analyse*, *manipulate* and *perform computations over* these new contracts, because they are described in terms of a fixed, well-understood set of primitives.

The major thrust of this chapter is to draw insights from the study of functional programming to illuminate the world of financial contracts. More specifically, our contributions are the following:

- We define a carefully-chosen set of combinators, and, through an extended sequence of examples in Haskell, we show that these combinators can indeed be used to describe a wide variety of contracts (Section 3).

- Our combinators can be used to *describe a contract*, but we also want to *process a contract*. Notably, we want to be able to *find the value of a contract*. In Section 4 we describe how to give an abstract *valuation semantics* to our combinators. A fundamentally-important property of this semantics is that it is *compositional*; that is, the value of a compound contract is given by combining the values of its sub-contracts.

- We sketch an implementation of our valuation semantics, using as an example a simple interest-rate model and its associated lattice (Section 5).

Stated in this way, our work sounds like a perfectly routine application of the idea of using a functional language to define a domain-specific combinator library, thereby effectively creating an application-specific programming language. Such languages have been defined for parsers, music, animations, hardware circuits, and many others [van Deursen et al., 2000]. However, from the standpoint of financial engineers, our language is truly radical: they acknowledge that the lack of a precise way to describe complex contracts is "the bane of our lives"[1].

It has taken us a long time to boil down the immense soup of actively-traded contracts into a reasonably small set of combinators; but once that is done, new vistas open up, because a single formal description can drive all manner of automated processes. For example, we can generate schedules for back-office contract execution, perform risk analysis optimisations, present contracts in new graphical ways (e.g. "time-lined" decision trees), provide animated simulations, and so on.

This chapter is addressed to a functional programming audience. We will introduce any financial jargon as we go.

c, d, u	Contract
o	Observable
t, s	Date, time
k	Currency
x	Dimensionless real value
p	Value process
v	Random variable

Figure 1: Notational conventions

2 Getting started

In this section we will informally introduce our notation for contracts, and show how we can build more complicated contracts out of simpler ones. We use the functional language Haskell throughout.

2.1 A simple contract

Consider the following simple contract, which might serve as a modest birthday present for Richard: "receive £100 on 13th February 2003". (A contract of this form is known to the industry as *zero-coupon discount bond*.) We can specify this contract, which we name c_1, thus:

$$c_1 :: Contract$$
$$c_1 = zcb\ t_1\ 100\ GBP$$

Figure 1 summarises the notational conventions we use throughout the chapter for variables, such as c_1 and t_1 in this definition.

The combinator *zcb* used in c_1's definition has the following type:

$$zcb :: Date \rightarrow Double \rightarrow Currency \rightarrow Contract$$

The first argument to *zcb* is a *Date*, which specifies a particular moment in time (i.e. both date and time). We provide a function, *mkDate*, that converts a date expressed as a friendly character string to a *Date*.

$$mkDate :: String \rightarrow Date$$

Now we can define Richard's birthdays in 2003 and 2004 like this:

$$t_1, t_2 :: Date$$

[1] The quote is from an informal response to a draft of our work.

$t_1 = mkDate$ "0800 GMT 13 Feb 2003"
$t_2 = mkDate$ "0800 GMT 13 Feb 2004"

2.2 Combining contracts

So *zcb* lets us build a simple contract. We can also combine contracts to make bigger contracts. A good example of such a combining form is *and*, whose type is:

$$and :: Contract \rightarrow Contract \rightarrow Contract$$

Using *and* we can define c_3, a contract that involves two payments:

$c_2, c_3 :: Contract$
$c_2 = zcb\ t_2\ 200\ GBP$
$c_3 = c_1\ `and`\ c_2$

That is, if Richard holds the contract c_3 he will benefit from a payment of £100 on his 2003 birthday, and another payment of £200 on his 2004 birthday.

In general, the contracts we can describe are between two parties, the *holder* of the contract, and the *counter-party*. Notwithstanding Biblical advice (Acts 20.35), by default the owner of a contract receives the payments, and makes the choices, specified in the contract. This situation can be reversed by the *give* combinator:

$$give :: Contract \rightarrow Contract$$

The contract *give c* is simply c with rights and obligations reversed, a statement we will make precise in Section 4.2. Indeed, when two parties agree on a contract, one acquires the contract c, and the other simultaneously acquires *give c*; each is the other's counter-party. For example, c_4 is a contract whose holder *receives* £100 at time t_1, and *pays* £200 at time t_2:

$$c_4 = c_1\ `and`\ give\ c_2$$

So far, each of our definitions has defined a new *contract* (c_1, c_2, etc.). It is also easy to define a new *combinator* (a function that builds a contract). For example, we could define *andGive* thus:

$andGive :: Contract \rightarrow Contract \rightarrow Contract$
$andGive\ c\ d = c\ `and`\ give\ d$

Now we can give an alternative definition of c_4 (which we built earlier):

$$c_4 = c_1\ `andGive`\ c_2$$

This ability to define new combinators, and *use them just as if they were built in*, is quite routine for functional programmers, but not for financial engineers.

zero :: *Contract*
: *zero* is a contract that has no rights and no obligations.

one :: *Currency* → *Contract*
: If you acquire (*one* k) you immediately receive one unit of the currency k.

give :: *Contract* → *Contract*
: To acquire (*give* c) is to acquire all c's rights as obligations, and vice versa. Note that for a bilateral contract q between parties A and B, A acquiring q implies that B acquires (*give* q).

and :: *Contract* → *Contract* → *Contract*
: If you acquire (c_1 `` `and` `` c_2), you immediately acquire *both* c_1 and c_2.

or :: *Contract* → *Contract* → *Contract*
: If you acquire (c_1 `` `or` `` c_2) you must immediately acquire your choice of *either* c_1 or c_2 (but not both).

cond :: *Obs Bool* → *Contract* → *Contract* → *Contract*
: If you acquire (*cond* b c_1 c_2), you acquire c_1 if the observable b is true *at the moment of acquisition*, and c_2 otherwise.

scale :: *Obs Double* → *Contract* → *Contract*
: If you acquire (*scale* o c), then you acquire c at the same moment, except that all the payments of c are multiplied by the value of the observable o *at the moment of acquisition*.

when :: *Obs Bool* → *Contract* → *Contract*
: If you acquire (*when* o c), you must acquire c as soon as observable o subsequently becomes *True*. It is therefore worthless in states where o will never again be *True*.

anytime :: *Obs Bool* → *Contract* → *Contract*
: Once you acquire (*anytime* o c), you *may* acquire c at any time the observable o is *True*. The compound contract is therefore worthless in states where o will never again be *True*.

until :: *Obs Bool* → *Contract* → *Contract*
: Once acquired, (*until* o c) is exactly like c except that it *must be abandoned* when observable o becomes *True*. In states in which o is *True*, the compound contract is therefore worthless, because it must be abandoned immediately.

Figure 2: Primitives for defining contracts

3 Building contracts

We have now completed our informal introduction. In this section we will give the full set of primitives, and show how a wide variety of other contracts can be built using them. For reference, Figure 2 gives the primitive combinators over contracts; we will introduce these primitives as we need them.

konst :: $a \rightarrow Obs\ a$
(*konst* x) is an observable that has value x at any time.

lift :: $(a \rightarrow b) \rightarrow Obs\ a \rightarrow Obs\ b$
(*lift* f o) is the observable whose value is the result of applying f to the value of the observable o.

$lift_2$ :: $(a \rightarrow b \rightarrow c) \rightarrow Obs\ a \rightarrow Obs\ b \rightarrow Obs\ c$
($lift_2$ f o_1 o_2) is the observable whose value is the result of applying f to the values of the observables o_1 and o_2.

date :: *Obs Date*
The value of the observable *date* at date s is just s.

instance Num a => Num (Obs a)
All numeric operations lift to the *Obs* type. The implementation is simple, using *lift* and $lift_2$.

Figure 3: Primitives over observables

3.1 Acquisition date and observables

Figure 2 gives an English-language, but quite precise, description of each combinator. To do so, it uses two concepts that we must introduce first: the notions of an acquisition date, and an observable.

Our language describes what a contract *is*. However, the *consequences for the holder* of the contract depend on the date at which the contract is acquired; that is, its *acquisition date*. (By "consequences for the holder" we mean the payments, rights and obligations that the contract confers on the holder of a contract.) For example, the contract "receive £100 on 1 Jan 2000 and receive £100 on 1 Jan 2001" is worth a lot less if acquired after 1 Jan 2000, because, by definition, any rights and obligations that fall due before the acquisition date are simply discarded.

The second fundamental concept is that of an *observable*. A real contract often depends on measurable quantities. For example, a contract might say "receive an amount in dollars equal to the noon Centigrade temperature in Los Angeles"; or "pay an amount in pounds sterling equal to the 3-month LIBOR spot rate[2] multiplied by 100". We use the term *observable* for an objective, but possibly *time-varying*, even perhaps *unknown at contracting time* quantity. By "objective" we mean that both parties to the contract will agree on the value of the observable at any particular time. The time of day, or the temperature in Los Angeles, can be objectively measured; but the value to a home-owner of insuring his house is subjective, and is not an observable. Observables are thus a different "kind of thing" from contracts, so we give them a different type:

date :: *Obs Date*

[2]The LIBOR spot rate is published daily in the financial press.

$tempInLA :: Obs\ Double$

In general, a value of type $Obs\ d$ represents a time-varying quantity of type d. We will often use the observable $date :: Obs\ Date$ in what follows.

Observables and their operations are, of course, reminiscent of Fran's *behaviours* [Elliott and Hudak, 1997]. Like Fran, we provide combinators for lifting functions to the observable level, *lift*, $lift_2$, etc. Figure 3 gives the primitive combinators over observables for reference; we will introduce them as we encounter them.

With these notions in hand, we now explore the combinators described in Figure 2.

3.2 Discount bonds

Earlier, we described the zero-coupon discount bond: "receive £100 at time t_1" (Section 2.1). At that time we assumed that *zcb* was a primitive combinator, but in fact it isn't. It is obtained by composing no fewer than four more primitive combinators. We begin with the *one* combinator:

$$c_5 = one\ GBP$$

Figure 2 gives a careful, albeit informal, definition of *one*: if you acquire (*one GBP*), you *immediately* receive £1.

But the bond we want pays £100, not £1. We use the combinator *scale* to "scale up" the contract, thus:

$$c_6 = scale\ (konst\ 100)\ c_5$$

To acquire (*scale o c*) is to acquire the contract c, but all the payments and receipts in c are multiplied by the value of observable o. In this case, we want to scale by the constant 100, so we use the combinator *konst* (from Figure 3) to lift the constant 100 to an observable whose value is always 100. You may wonder why *scale* takes an observable, rather than simply a constant; we discuss that in the next sub-section.

We are not finished with our zero-coupon bond, however. c_6 pays the right *amount*, but at the wrong *time*: it pays at the moment of acquisition, whereas it should pay at date t_1 and no earlier. To obtain this effect we use the *when* combinator (Figure 2):

$$c_7 = when\ (at\ t_1)\ c_6$$

If you acquire the contract (*when o c*), where o is a boolean observable, then nothing happens until o becomes *True*; when that happens, you immediately acquire c. So (*at* t_1) should be a boolean observable that becomes *True* at time t_1. We can define *at* like this:

$$at :: Date \rightarrow Obs\ Bool$$
$$at\ t = lift_2\ (==)\ date\ (konst\ t) \quad \text{— } True \text{ when } (date == t)$$

The $lift_2$ takes (==), which compares two *Date*s and retuns a *Bool*, to a function that takes two *Obs Date* arguments and returns an *Obs Bool* (Figure 3). We use *konst* again, to lift the date t to an observable.

Notice that if we acquire c_7 after t_1 we get nothing, because $(at\ t_1)$ is *True at* the time t_1, but then becomes *False* and stays *False* forever.

Putting all this together, we can finally define *zcb* correctly:

$$\begin{aligned} &zcb :: Date \to Double \to Currency \to Contract \\ &zcb\ t\ x\ k\ =\ when\ (at\ t)\ (scale\ (konst\ x)\ (one\ k)) \end{aligned}$$

These definitions of *zcb* and *at* effectively extend our repertoire of combinators, just as *andGive* did in Section 2.2, only more usefully. We will continually extend our library of combinators in this way.

Why did we go to the trouble of defining *zcb* in terms of four combinators, rather than making it primitive? Because it turns out that *scale*, *when*, *lift*, $lift_2$, and *one* are all independently useful. Each embodies a distinct piece of functionality, and by separating them we significantly simplify the semantics and enrich the algebra of contracts (Section 4). The combinators we present are the result of an extended, iterative process of refinement, leading to an interlocking set of decisions — programming language designers will be quite familiar with this process.

3.3 Observables and scaling

In the previous section we used *scale* to scale a contract by a fixed quantity. But, as we saw, *scale* scales a contract by an *observable*, that is, by a time-varying, maybe unknown in advance, value:

$$scale :: Obs\ Double \to Contract \to Contract$$

Why do we want this generality? So-called "weather derivatives" are a good example; a holiday resort might insure against bad weather by buying a contract that pays out an amount depending on the amount of rain:

$$c_8\ =\ scale\ rainInCyprus\ (one\ USD)$$

Here we assume that *rainInCyprus* is a primitive observable:

$$\begin{aligned} &rainInCyprus :: Obs\ Double \quad \text{— Rainfall over a 24 hr period in} \\ &\qquad\qquad\qquad\qquad\qquad\quad \text{— Cyprus, measured in centimetres} \end{aligned}$$

If you acquire this contract you receive an amount in dollars equal to the 24-hr rainfall in Cyprus, expressed in centimetres. Again, we have to be very precise in our definitions. Exactly when is the rainfall sampled? Answer (in Figure 2): when you acquire $(scale\ o\ c)$ you immediately acquire c, scaling all the payments and receipts in c by the value of the observable o *sampled at the moment of acquisition.* So we sample the observable at a single, well-defined moment (the acquisition date) and then use that single number to scale the subsequent payments and receipts in

c. The *when* combinator is often used to define precisely an acquisition date, but we will see other possibilities in the next sections.

It is not long before one wants to perform arithmetic on observables. For example, suppose you want to receive \$1,000 for every centimetre of rainfall over 7cm. We can easily do that, using the $lift_2$ combinator we used earlier:

$$\begin{aligned} c_9 \;=\; & scale\ (lift_2\ (*) \\ & \qquad (lift_2\ (-)\ rainInCyprus\ (konst\ 7)) \\ & \qquad (konst\ 1000)) \\ & (one\ USD) \end{aligned}$$

This is all rather clumsy, but fortuntely Haskell's type-class overloading mechanism allows us to use the arithmetic operators $-$ and $*$ directly on observables, thus:

$$c_9 \;=\; scale\ ((rainInCyprus\ -\ 7)\ *\ 1000)\ (one\ USD)$$

To achieve this, we simply need to make *Obs* into an instance of *Num*, thus:

$$\begin{aligned} & instance\ Num\ a\ =>\ Num\ (Obs\ a)\ where \\ & fromInteger\ i\ =\ konst\ (fromInteger\ i) \\ & (+)\ =\ lift_2\ (+) \\ & (-)\ =\ lift_2\ (-) \\ & ...etc... \end{aligned}$$

Unfortunately, Haskell's type system does not work quite so smoothly for relational operators, such as $(<)$. To reduce notational clutter, we will define a family of relational operators, thus:

$$\begin{aligned} & (\%<),\ (\%<=),\ (\%=),\ (\%>=),\ (\%>)\ ::\ Ord\ a\ =>\ Obs\ a\ \rightarrow\ Obs\ a\ \rightarrow\ Obs\ Bool \\ & (\%<)\ =\ lift_2\ (<) \\ & (\%<=)\ =\ lift_2\ (<=)\ ...etc... \end{aligned}$$

The *scale* combinator allows an observable to control the size, but not the character, of a contract. The *cond* combinator allows an observable to select which of two contracts is acquired:

$$cond\ ::\ Obs\ Bool\ \rightarrow\ Contract\ \rightarrow\ Contract$$

For example, the contract

$$c_{10}\ =\ cond\ (rainInCyrus\ \%>\ 10)\ (one\ GBP)\ (one\ USD)$$

If the rainfall in Cyprus, sampled on the date when c_{10} is acquired, is more than 10cm, you receive £1, otherwise you receive \$1.

3.4 Option contracts

Much of the subtlety in financial contracts arises because the participants can exercise *choices*. We encapsulate choice in three primitive combinators, *cond*, *or*

and *anytime*. The first two allow one to choose *which* of two contracts to acquire, while the last allows one to choose *when* to acquire a contract.

First, we consider the choice between two contracts:

$$or :: Contract \rightarrow Contract \rightarrow Contract$$

When you acquire the contract (c_1 `` `or` `` c_2), you must immediately acquire either c_1 or c_2 (but not both). For example, the contract

$$zcb\ t_1\ 100\ GBP\ \text{`}or\text{`}\ zcb\ t_2\ 110\ GBP$$

gives the holder the right, to choose immediately to receive either £100 at t_1, or £110 at t_2.

A so-called *European option* gives the right to choose, at a particular date, whether or not to acquire an "underlying" contract:

$$european :: Date \rightarrow Contract \rightarrow Contract$$

For example, consider the contract c_5:

$$\begin{aligned} c_{11} = {} & european\ (date\ \text{“}1200GMT\ 24\ Apr\ 2003\text{”})\ (\\ & zcb\ (mkDate\ \text{“}1200GMT\ 12\ May\ 2003\text{”})\ 0.4\ GBP\ \text{`}and\text{`} \\ & zcb\ (mkDate\ \text{“}1200GMT\ 12\ May\ 2004\text{”})\ 9.3\ GBP\ \text{`}and\text{`} \\ & zcb\ (mkDate\ \text{“}1200GMT\ 12\ May\ 2005\text{”})\ 109.3\ GBP\ \text{`}and\text{`} \\ & give\ (zcb\ (mkDate\ \text{“}1200GMT\ 26\ Apr\ 2003\text{”})\ 100\ GBP) \\ &) \end{aligned}$$

This contract gives the right to choose, on 24 Apr 2003, whether or not to acquire an underlying contract consisting of three receipts and one payment. In the financial industry, this kind of contract is described as a "call on a coupon bond". As with *zcb*, we define *european* in terms of simpler elements:

$$\begin{aligned} & european :: Date \rightarrow Contract \rightarrow Contract \\ & european\ t\ u = when\ (at\ t)\ (u\ \text{`}or\text{`}\ zero) \end{aligned}$$

You can read this definition as follows:

- The primitive contract *zero* has no rights or obligations (see Figure 2).
- The contract (u `` `or` `` *zero*) expresses the choice between acquiring u and acquiring nothing.
- The *when* (*at* t) construct means that the choice must be made exactly at date t. If you acquire the European contract after t, you get nothing.

The *or* combinator lets us choose *which* of two contracts to acquire. Let us now consider the choice of *when* to acquire a contract:

$$anytime :: Obs\ Bool \rightarrow Contract \rightarrow Contract$$

Acquiring the contract *anytime o u* gives the right to acquire the "underlying" contract u at any time that the observable o is True.

An *American option* offers more flexibility than a European option. Typically, an American option confers the right to acquire an underlying contract *at any time between two dates* (or not to do so at all).

$$american\ (t_1, t_2)\ u\ =\ anytime\ (between\ t_1\ t_2)\ u$$

Here we use another simple combinator for observables:

```
between :: Date → Date → Obs Bool
  — True on dates in the specified window
between t1 t2 = lift2 (&&) (date %>= t1) (date %<= t2)
```

3.5 Limit contracts

The last form of contract we define is a "limit contract". Such contracts say "such-and-such happens unless interest rates go above 6%", or "unless the temperature falls below freezing". To accomodate such contracts we need a way to *abandon* a contract, and that is the purpose of the *until* combinator. For example:

$$c_{12}\ =\ until\ (interestRate\ \%>\ konst\ 6)\ (american\ (t_1, t_2)\ c)$$

When you acquire c_{12} you immediately acquire the underlying American option, but as soon as interest rates go above 6% you must abandon the option, whether or not you have by then exercised the option and acquired c. If you have exercised the option, and c has paid out some money, you get to keep that, but you abandon any future benefits of c.

3.6 Summary

We have now given the flavour of our approach to defining contracts. The combinators we have defined so far are not enough to describe all the contracts that are actively traded, and we are extending the set in ongoing work. However, our main conclusions are unaffected:

- Financial contracts can be described in a purely declarative way.
- A huge variety of contracts can be described in terms of a small number of combinators.

Identifying the "right" primitive combinators is quite a challenge. For example, it was a breakthrough to identify and separate the two forms of choice, *or* and *anytime*, and encapsulate those choices (and nothing else) in two combinators. Another breakthrough was to use boolean observables to describe the "region" in which one can acquire a contract.

4 Valuation

We now have at our disposal a rich language for *describing* financial contracts. This is already useful for communicating between people — the industry lacks any such precise notation. But in addition, a precise description lends itself to automatic processing of various sorts. From a single contract description we may hope to generate legal paperwork, pictures, schedules and more besides. The most immediate question one might ask about a contract is, however, *what is it worth?* That is, *what would I pay to own the contract?* It is to this question that we now turn.

We will express contract valuation in two "layers":

Abstract valuation semantics. First, we will show how to translate an arbitrary contract, written in our language, into a *value process*, together with a handful of operations over these processes. These processes correspond directly to the mathematical and stochastic machinery used by financial experts.

Concrete implementation. A process is an abstract mathematical value. To make a computer calculate with processes we have to represent them somehow — this is the step from abstract semantics to concrete implementation. An implementation will consist of a *financial model*, together with some discrete *numerical method*. A tremendous number of different financial models are used today (e.g. Black-Scholes, Ho-Lee, etc.); but only three families of numerical methods are widely used in industry: partial differential equations [Willmot et al., 1993], Monte Carlo [Boyle et al., 1997] and lattice methods [Cox et al., 1979].

This approach is strongly reminiscent of the way in which a compiler is typically structured. The program is first translated into a low-level but machine-independent intermediate language; many optimisations are applied at this level; and then the program is further translated into the instruction set for the desired processor (Pentium, Sparc, or whatever).

In a similar way, we can transform a contract into a value process, apply meaning-preserving optimising transformations to this intermediate representation, before computing a value for the process. This latter step can be done interpretatively, or one could imagine generating specialised code that, when run, would perform the valuation.

Indeed, our abstract semantics serves as our reference model for what it means for two contracts to be "the same". For example, here are two claims:

$$\begin{array}{rcl} c_1\ `and`\ (c_2\ `or`\ c_3) & = & (c_1\ `and`\ c_2)\ `or`\ (c_1\ `and`\ c_3) \\ give\ (c_1\ `or`\ c_2) & = & give\ c_1\ `or`\ give\ c_2 \end{array}$$

In fact, the first is true, and the second is not, but how do we know for sure? Answer: we compare their valuation semantics, as we shall see in Section 4.6.

$$\mathcal{E}_{\mathrm{k}}[\![\cdot]\!] : \mathit{Contract} \rightarrow \mathcal{PR}\ \mathbf{R}$$

$$\begin{array}{lrcl}
\text{(E1)} & \mathcal{E}_{\mathrm{k}}[\![\mathit{zero}]\!] & = & \mathcal{K}(0) \\
\text{(E2)} & \mathcal{E}_{\mathrm{k}}[\![\mathit{one}\ k_2]\!] & = & \mathit{exch}_k(k_2) \\
\text{(E3)} & \mathcal{E}_{\mathrm{k}}[\![\mathit{give}\ c]\!] & = & -\ \mathcal{E}_{\mathrm{k}}[\![c]\!] \\
\text{(E4)} & \mathcal{E}_{\mathrm{k}}[\![o\ \text{`}\mathit{scale}\text{'}\ c]\!] & = & \mathcal{V}[\![o]\!]\ *\ \mathcal{E}_{\mathrm{k}}[\![c]\!] \\
\text{(E5)} & \mathcal{E}_{\mathrm{k}}[\![c_1\ \text{`}\mathit{and}\text{'}\ c_2]\!] & = & \mathcal{E}_{\mathrm{k}}[\![c_1]\!]\ +\ \mathcal{E}_{\mathrm{k}}[\![c_2]\!] \\
\text{(E6)} & \mathcal{E}_{\mathrm{k}}[\![c_1\ \text{`}\mathit{or}\text{'}\ c_2]\!] & = & \mathit{max}(\mathcal{E}_{\mathrm{k}}[\![c_1]\!],\ \mathcal{E}_{\mathrm{k}}[\![c_2]\!]) \\
\text{(E7)} & \mathcal{E}_{\mathrm{k}}[\![\mathit{cond}\ o\ c_1\ c_2]\!] & = & \mathit{if}(\mathcal{V}[\![o]\!],\ \mathcal{E}_{\mathrm{k}}[\![c_1]\!],\ \mathcal{E}_{\mathrm{k}}[\![c_2]\!]) \\
\text{(E8)} & \mathcal{E}_{\mathrm{k}}[\![\mathit{when}\ o\ c]\!] & = & \mathit{disc}_k(\mathcal{V}[\![o]\!], \mathcal{E}_{\mathrm{k}}[\![c]\!]) \\
\text{(E9)} & \mathcal{E}_{\mathrm{k}}[\![\mathit{anytime}\ o\ c]\!] & = & \mathit{snell}_k(\mathcal{V}[\![o]\!], \mathcal{E}_{\mathrm{k}}[\![c]\!]) \\
\text{(E10)} & \mathcal{E}_{\mathrm{k}}[\![\mathit{until}\ o\ c]\!] & = & \mathit{absorb}_k(\mathcal{V}[\![o]\!], \mathcal{E}_{\mathrm{k}}[\![c]\!])
\end{array}$$

Figure 4: Compositional valuation semantics for contracts

4.1 Value processes

Definition 1 (Value process.) A *value process*, p, over type a, is a (total) function from time to a random variable of type a. The random variable $p(t)$ describes the possible values for p at time t. We write the informal type definition

$$\mathcal{PR}\ a = \mathit{Date} \rightarrow \mathcal{RV}\ a$$

□

(We use caligraphic font for types at the semantic level.) Because we need to work with different processes defined on the same "underlying space" (technically, filtration), such a value process is more precisely described as an *adapted stochastic process, given a filtration*. Such processes come equipped with a sophisticated mathematical theory [Revuz and Yor, 1991, Musiela and Rutkowski, 1997], but it is unlikely to be familiar to computer scientists, so we only present informal, intuitive notions. We usually abbreviate "value process" to simply "process". Be warned, though: "process" and "variable" mean quite different things to their conventional computer science meanings.

Both contracts and observables are modeled as processes. The underlying intuitions are as follows:

- The value process for an observable o maps a time t to a random variable describing the possible values of o at t. For example, the value process for the observable "IBM stock price in US$" is a function that maps a time to a real-valued random variable that describes the possible values of IBM's stock price in US$.

$$
\begin{array}{rcl}
\mathcal{V}[\![\cdot]\!] : \mathit{Obs}\ a & \to & \mathcal{PR}\ a \\
\\
\mathcal{V}[\![\mathit{konst}\ x]\!] & = & \mathcal{K}(x) \\
\mathcal{V}[\![\mathit{date}]\!] & = & \mathit{date} \\
\mathcal{V}[\![\mathit{lift}\ f\ o]\!] & = & \mathit{lift}(f,\ \mathcal{V}[\![o]\!]) \\
\mathcal{V}[\![\mathit{lift}_2\ f\ o_1\ o_2]\!] & = & \mathit{lift}_2(f,\ \mathcal{V}[\![o_1]\!],\ \mathcal{V}[\![o_2]\!])
\end{array}
$$

Figure 5: Valuation semantics for observables

The following primitives are independent of the valuation model

$\mathcal{K} : a \to \mathcal{PR}\ a$
: The process $\mathcal{K}(x)$ is defined at all times to have value x.

$\mathit{date} : \mathcal{PR}\ \mathit{Date}$
: The process *date* has as its value the date.

$\mathit{cond} : \mathcal{PR}\ \mathit{Bools} \to \mathcal{PR}\ a \to \mathcal{PR}\ a \to \mathcal{PR}\ a$
: Conditional choice between the latter two processes, based on the first.

$\mathit{lift} : (a \to b) \to \mathcal{PR}\ a \to \mathcal{PR}\ b$
: Apply the specified function to the argument process point-wise.

$\mathit{lift}_2 : (a \to b \to c) \to \mathcal{PR}\ a \to \mathcal{PR}\ b \to \mathcal{PR}\ c$
: Combine the two argument processes point-wise with the specified function.

$+, *, \ldots : \mathcal{PR}\ \mathbf{R} \to \mathcal{PR}\ \mathbf{R} \to \mathcal{PR}\ \mathbf{R}$
: Add, or multiply (etc.) the two processes. Equivalent to $\mathit{lift}_2(+)$, etc.

Figure 6: Process primitives

- The value process for a contract c, expressed in currency k is a function from a time, t, to a random variable describing the value, in currency k, of *acquiring* the contract c *at time* t.

These intuitions are essential to understand the rest of the chapter.

4.2 From contracts to processes

How, then, are we to go from contracts and observables to processes? Figure 4 gives the complete translation from contracts to processes, while Figure 5 does the same for observables. These Figures do not look very impressive, but that is

The following primitives are dependent on the particular model

$exch_k(\cdot) : Currency \to \mathcal{PR}\ \mathbf{R}$
: $exch_{k_1}(k_2)$ is a real-valued process representing the value of one unit of k_2, expressed in currency k_1. This is simply the process representing the quoted exchange rate between the currencies.

$disc_k(\cdot,\cdot) : \mathcal{PR}\ \mathbf{B} \times \mathcal{PR}\ \mathbf{R} \to \mathcal{PR}\ \mathbf{R}$
: Given a boolean-valued process o, the primitive $disc_k(o,p)$ transforms the real-valued process p, expressed in currency k, into another real-valued process. In states where o is *True*, the result is the same as p; elsewhere, the result is its "fair" equivalent stochastic value process in the same currency k.

$snell_k(\cdot,\cdot) : \mathcal{PR}\ \mathbf{B} \times \mathcal{PR}\ \mathbf{R} \to \mathcal{PR}\ \mathbf{R}$
: The primitive $snell_k(o,p)$ calculates the Snell envelope of its argument p, under observable o. It uses the probability measure associated with the currency k.

$absorb_k(\cdot,\cdot) : \mathcal{PR}\ \mathbf{B} \times \mathcal{PR}\ \mathbf{R} \to \mathcal{PR}\ \mathbf{R}$
: Given a boolean-valued process o, the primitive $absorb_k(o,p)$ transforms the real-valued process p, expressed in currency k, into another real-valued process. For any state, the result is the expected value of receiving p's value if the region o will never be *True*, and receiving zero in the contrary. In states where o is *True*, the result is therefore zero.

Figure 7: Model primitives

the whole point! Everything so far has been leading up to this point; our entire design is organised around the desire to give a simple, tractable, modular valuation semantics. Let us look at Figure 4 in more detail.

The function $\mathcal{E}_{\mathbf{k}}[\![c]\!]$ takes a contract, c, and maps it to a process describing, for each moment in time, the value in currency k of acquiring c at that moment. For example, the equation for *give* (E3) says that the value process for *give* c is simply the negation of $\mathcal{E}_{\mathbf{k}}[\![c]\!]$, the value process for c. Aha! What does "negation" mean? Clearly, we need not only the notion of a value process, but also a collection of operations over these processes. Negating a process is one such operation; the negation of a process p is simply a function that maps each time, t, to the negation of $p(t)$. It is an absolutely straightforward exercise to "lift" all operations on real numbers to operate point-wise on processes. (This, in turn, requires us to negate a random variable, but doing so is simple.) We will need a number of other operations over processes. They are summarised in Figures 6 and 7, but we will introduce each one as we need it.

Next, consider equation (E5). The *and* of two contracts is modeled simply by

taking the sum of their two value processes. Equation (E6) does the same for the *or* combinator. Again, by design, the combinator maps to a simple mathematical operation, max.

Equation (E4) is nice and simple. To scale a contract c by a time-varying observable o, we simply multiply the value process for the contract $\mathcal{E}_{\mathtt{k}}[\![c]\!]$ by the value process for the observable — remember that we are modeling each observable by a value process. We express the latter as $\mathcal{V}[\![o]\!]$, defined in Figure 5 in a very similar fashion to $\mathcal{E}_{\mathtt{k}}[\![c]\!]$. At first this seems odd: how can we scale point-wise, when the scaling applies to *future* payments and receipts in c? Recall that the value process for c at a time t gives the value of acquiring c at t. Well, if this value is v then the value of acquiring the same contract with all payments and receipts scaled by x is certainly $v * x$. Our definition of *scale* in Figure 2 was in fact driven directly by our desire to express its semantics in a simple way. Simple semantics gives rise to simple algebraic properties (Section 4.6).

The equations for *zero* and *cond* are also easy. Equation (E1) delivers the constant zero process, while Equation (E7) uses the underlying conditional.

4.3 Exchange rates

The operations over value processes defined in Figure 6 are generic – they are unrelated to a particular financial model. But we can't get away with that forever. The primitives in Figure 7 are specific to financial contracts, and they are used in the remaining equations of Figure 4.

Consider equation (E2) in Figure 4. It says that the value process for one unit of currency k_2, expressed in currency k, is simply the exchange-rate process between k_2 and k namely $exch_k(k_2)$ (Figure 7). Where do we get these exchange-rate processes from? When we come to implementation, we will need some (numerical) assumption about future evolution of exchange rates, but for now it suffices to treat the exchange-rate processes as primitives. However, there are important relationships between them! Notably:

$$\begin{array}{lrcl} \text{(A1)} & exch_k(k) & = & \mathcal{K}(1) \\ \text{(A2)} & exch_{k_2}(k_1) * exch_{k_3}(k_2) & = & exch_{k_3}(k_1) \end{array}$$

That is, the exchange-rate process between a currency and itself is everywhere unity; and it makes no difference whether we convert k_1 directly into k_3 or whether we go via some intermediate currency k_2. These are particular cases of *no-arbitrage* conditions[3].

You might also wonder what has become of the bid-offer spread encountered by every traveller at the foreign-exchange counter. In order to keep things technically tractable, finance theory assumes most of the time the absence of any spreads: one typically first computes a "fair" price, before finally adding a profit margin. It is the latter which gives rise to the spread, but our modeling applies only to the former.

[3] A *no-arbitrage* condition is one that excludes a *risk-free* opportunity to earn money. If such an opportunity were to exist, everyone would take it, and the opportunity would soon go away!

4.4 Interest rates

Next, consider equation (E8). The contract (*when o c*) acquires the underlying contract c as soon as the observable o becomes *True*, or immediately if o is *True* at the moment (*when o c*) is acquired. In states where o is *True*, the value of (*when o c*) is therefore the same as the value of c. What is the value of (*when o c*) in states where o is *False*? To answer that question we need a specification of future evolution of interest rates, that is an interest-rate model.

Let's consider a concrete example:

$$c = \mathit{when}\ (\mathit{at}\ t)\ (\mathit{scale}\ (\mathit{konst}\ 10)\ (\mathit{one}\ \mathit{GBP}))$$

where t is one year from today. The underlying contract pays out £10 immediately when it is acquired; the *when* acquires it at t. So the value of c at t is just £10. Before t, though, it is not worth as much. If I expect interest rates to average[4] (say) 10% over the next year, a fair price for c today would be about £9.

Just as the primitive *exch* encapsulates assumptions about future exchange-rate evolution, so the primitive $\mathit{disc}_k(o,p)$ encapsulates an interest-rate evolution (Figure 7). Here o is a boolean-valued process that defines a region that we call the *acquisition region*. Inside this region — in states where o has value *True* — $\mathit{disc}_k(o,p)$ is equal to p. But elsewhere the value of $\mathit{disc}_k(o,p)$ is obtained by computing the "discounted expected value" of p. For example, suppose p is everywhere equal to 100, and o is *True* at all times greater than 1 Jan 2003. Then the value of $\mathit{disc}_{\pounds}(o,p)$ at 1 Feb 2003 is 100. However, its value at 1 Jan 2002 depends on sterling interest rates: we must compute the value at 1 Jan 2002 of acquiring £100 on 1 Jan 2003.

Like *exch*, there are some properties that any no-arbitrage financial model should satisfy. Notably:

$$\begin{array}{lrcl}
(A3) & \mathit{disc}_k(\mathcal{K}(\mathit{True}),p) & = & p \\
(A4) & \mathit{exch}_{k_1}(k_2) * \mathit{disc}_{k_2}(o,p) & = & \mathit{disc}_{k_1}(o, \mathit{exch}_{k_1}(k_2) * p) \\
(A5) & \mathit{disc}_k(o,p_1+p_2) & = & \mathit{disc}_k(o,p_1) + \mathit{disc}_k(o,p_2)
\end{array}$$

The first equation says that *disc* should be the identity when the acquisition region is the entire space; the second says that the interest-rate evolution of different currencies should be compatible with the assumption of evolution of exchange rates. The third[5] is often used in a right-to-left direction as an optimisation: rather than perform discounting on two random variables separately, and then add the resulting processes, it is faster to add the random variables and then discount the result. Just as in an optimising compiler, we may use identities like these to transform (the meaning of) our contract into a form that is faster to execute.

One has to be careful, though. Here is a plausible property that does *not* hold:

$$\mathit{disc}_k(o, \mathit{max}(p_1, p_2)) = \mathit{max}(\mathit{disc}_k(o,p_1), \mathit{disc}_k(o,p_2))$$

[4] For the associated *risk-neutral* probability, but we will not go in these financial details here.

[5] The financially educated reader should note that we assume here implicitly what are called *complete* markets.

It is plausible because it would hold if p_1, p_2 were single numbers and *disc* were a simple multiplicative factor. But p_1 and p_2 are random processes, and the property is false.

Equation (E9) uses the *snell* operator to give the meaning of *anytime*. This operator is mathematically subtle, but it has a simple characterisation: $snell_k(o, p)$ is the smallest process q (under an ordering relation we mention briefly at the end of Section 4.6) such that

$$\forall o'.\ (o \Rightarrow o') \ \Rightarrow\ q \geq snell_k(o', q)$$

That is, an American option is the least upper bound of any of the deterministic acquisition choices specified by o', where o' is a sub-region of o.

4.5 Observables

We can only value contracts over observables that we can model. For example, we can only value a contract involving the temperature in Los Angeles if we have a model of the temperature in Los Angeles. Some such observables clearly require separate models. Others, such as the LIBOR rate and the price of futures, can incestuously be modeled as the value of particular contracts. We omit all the details here; Figure 5 gives the semantics only for the simplest observables. This is not unrealistic, however. One can write a large range of contracts with our contract combinators and only these simple observables.

4.6 Reasoning about contracts

Now we are ready to use our semantics to answer the questions we posed at the beginning of Section 4. First, is this equation valid?

$$c_1\ `and`\ (c_2\ `or`\ c_3)\ =\ (c_1\ `and`\ c_2)\ `or`\ (c_1\ `and`\ c_3)$$

By taking the meaning of the left hand side, we get

$$\begin{array}{rcl} \mathcal{E}_{\mathrm{k}}[\![LHS]\!] & = & c_1 + (max(c_2, c_3)) \\ & = & max(c_1 + c_2, c_1 + c_3)) \\ & = & \mathcal{E}_{\mathrm{k}}[\![RHS]\!] \end{array}$$

where $c_1 = \mathcal{E}_{\mathrm{k}}[\![c_1]\!]$ etc. In a similar way, we can argue this plausible equation is false:

$$give\ (c_1\ `or`\ c_2)\ \stackrel{?}{=}\ give\ c_1\ `or`\ give\ c_2$$

The proof is routine, but its core is the observation that

$$-max(a, b) \neq max(-a, -b)$$

Back in the real world, the point is that the left hand side gives the choice to the counter-party, whereas in the right hand side the choice is made by the holder of the contract.

Our combinators satisfy a rich set of equalities, such as that given for *or* and *and* above. Some of these equalities have side conditions. For example:

$$scale\ o\ (c_1\ `or`\ c_2) = scale\ o\ c_1\ `or`\ scale\ o\ c_2$$

holds only if $o \geq 0$, for exactly the same reason that *give* does not commute with *or*. Hang on! What does it mean to say that "$o \geq 0$"? We mean that o is positive for all time. More generally, as well as equalities between contracts, there is a simple notion of ordering between contracts, $c_1 \geq c_2$, pronounced "c_1 *dominates* c_2": $c_1 \geq c_2$ if it is never at any time preferable to acquire c_2 than to acquire c_1. The ordering is defined by simply comparing the value processes of the two contracts; that is, $c_1 \geq c_2$ iff $\mathcal{E}[\![c_1]\!] \geq \mathcal{E}[\![c_2]\!]$. One value process is greater than another iff it is greater in all states of the world. (The ordering is only a partial one.)

Equalities, such as the ones given above, can be used as optimising transformations in a valuation engine. A "contract compiler" can use these identities to transform a contract, expressed in the intermediate language of value processes (see the introduction to Section 4), into a form that can be valued more efficiently.

4.7 Summary

This completes our description of the abstract valuation semantics. From a programming-language point of view, everything is quite routine, including our proofs. But we stress that it is most unusual to find formal proofs in the finance industry at this level of abstraction. We have named and tamed the complicated primitives (*disc*, *exch*, etc.): the laws they must satisfy give us a way to prove identities about contracts without having to understand much about random variables. The mathematical details are arcane, believe us!

5 Implementation

Our valuation semantics is not only an abstract beast. We can also regard Figures 4 and 5 as a translation from our contract language into a lower-level language of processes, whose combinators are the primitives of Figures 6 and 7. Then we can optimise the process-level description, using identities such as (A1)-(A5). (We use dozens of such identities as transformation rules.) Finally, all we need to do is to implement the process-level primitives, and we will be able to value an arbitrary contract.

The key decision is, of course, how we implement a value process. A value process has to represent *uncertainty about the future* in an explicit way. There are numerous ways to model this uncertainty. For the sake of concreteness, we will simply pick the Ho and Lee model, and use a lattice method to evaluate contracts with it [Ho and Lee, 1986]. We choose this model and numerical method for their technical simplicity and historical importance, but much of this section is also applicable to other models (e.g. Black-Derman-Toy). Changing the numerical

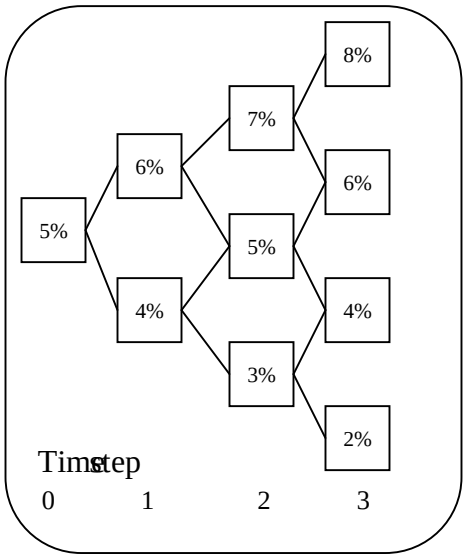

Figure 8: A short term interest-rate evolution

method (e.g. to Monte Carlo) would entail bigger changes, but *nothing in our language or its semantics (Sections 1-4) would be affected*. Indeed, it is entirely possible to use different numerical methods for different parts of a single contract.

5.1 An interest-rate model

In the typical Ho and Lee numerical scheme, the interest-rate evolution is represented by a lattice (or "recombining tree"), as depicted in Figure 8. Each column of the tree represents a discrete time step, and time increases from left to right. Time zero represents "now". As usual with discrete models, there is an issue of how long a time step will be; we won't discuss that further here, but we note in passing that the time steps need not be of uniform size.

At each node of the tree is associated a one-period short-term interest rate, shortly denominated the interest rate from now on. We know today's interest rate, so the first column in the tree has just one element. However, there is some uncertainty of what interest rates will evolve to by the end of the first time step. This is expressed by having two interest-rate values in the second column; the idea is that the interest rate will evolve to one of these two values with equal probability. In the third time step, the rates split again, *but the down/up path joins the up/down path*, so there are only three rates in the third column, not four. This is why the structure is called a lattice; it makes the whole scheme computationally feasible by giving only a linear growth in the width of the tree with time. Of course, the tree is only a discrete approximation of a continuous process; its recombining nature is just a choice for efficiency reasons. We write R_t for the vector of rates in time-

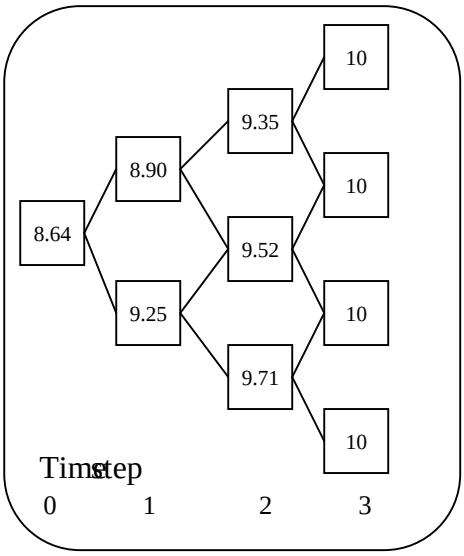

Figure 9: A Ho and Lee valuation lattice

step t, and $R_{t,i}$ for the i'th member of that vector, starting with 0 at the bottom. Thus, for example, $R_{2,1} = 5\%$. The actual numbers in Figure 8 are unrealistically regular: in more elaborate interest-rate models, they will not be evenly spaced but only monotonically distributed in each column.

5.2 Value processes

So much for the interest-rate model. A value process is modeled by a lattice of exactly the same shape as the interest-rate evolution, except that we have a *value* at each node instead of an *interest rate*. Figure 9 shows the value process tree for our favourite zero-coupon bond

$$c_7 \;=\; \mathit{when}\ (\mathit{at}\ t)\ (\mathit{scale}\ (\mathit{konst}\ 10)\ (\mathit{one}\ \mathit{GBP}))$$

evaluated in pounds sterling (GBP). Using our valuation semantics we have

$$\begin{aligned}\mathcal{E}_{GBP}[\![c_7]\!] &= \mathit{disc}_{\pounds}(\mathit{date} = t, \mathcal{K}(10) * \mathit{exch}_{\pounds}(\pounds)) \\ &= \mathit{disc}_{\pounds}(\mathit{date} = t, \mathcal{K}(10) * \mathcal{K}(1)) \\ &= \mathit{disc}_{\pounds}(\mathit{date} = t, \mathcal{K}(10))\end{aligned}$$

In the figure, we assume that the time t is time step 3. At step 3, therefore, the value of the contract c is certainly 10 at all nodes, because c unconditionally delivers £10 at that time. At time step 2, however, we must discount the £10 by the interest rate appropriate to that time step. We compute the value at each node of time-step 2 by averaging the two values in its successors, and then discounting

the average value back one time step using the interest rate associated to that node[6]. Using the same notation for the value tree V as we used for the rate model R, we get the equation:

$$V_{t,i} = \frac{V_{t+1,i} + V_{t+1,i+1}}{2(1 + R_{t,i}\Delta t)}$$

where Δt is the size of the time step. Using this equation we can fill in the rest of the values in the tree, as we have done in Figure 9. The value in time step 0 is the current value of the contract, in pounds sterling (i.e. £8.64).

In short, a lattice implementation works as follows:

- A value process is represented by a lattice, in which each column is a discrete representation of a random variable. The value in each node is one of the possible values the variable can take, and in our very simple setting the number of paths from the root to the node is proportional to the probability that the variable will take that value. We will say a bit more about how to represent such a tree in the next subsection.

- The generic operations, in Figure 6, are easy to implement. $\mathcal{K}(x)$ is a value process that is everywhere equal to x. *date* is a process in which the values in a particular column are all equal to that column's date. $\mathit{lift}(f, p)$ applies f to p point-wise; $\mathit{lift}_2(f, p_1, p_2)$ "zips together" p_1 and p_2, combining corresponding values point-wise with f.

- The model-specific operations of Figure 7 are a bit harder. We have described how to implement *disc*, which uses the interest-rate model. *exch* is actually rather easier (multiply the value process point-wise by a process representing the exchange rate). The *snell* and *absorb* primitives take a bit more work, and we do not describe them in detail here. Nevertheless, to give the idea, here is a possible implementation for *snell*: take the final column of the tree, discount it back one time step, take the maximum of that column with the corresponding column of the original tree, and then repeat that process all the way back to the root.

The remaining high-level question is: in the (big) set of possible interest-rate models, what is a "good" model? The answer is rather incestuous. A candidate interest-rate model should price correctly those contracts that are widely traded: one can simply look up the current market prices for them, and compare them with the calculated results. So we look for and later adjust the interest-rate model until it fits the market data for these simple contracts. Now we are ready to use the model to compute prices for more exotic contracts. The entire market is a gigantic feedback system, and active research studies the problem of its stability.

[6] For evident presentation reasons, we don't care about the fact that the Ho and Lee model is a member of a class of models that admit in fact a *closed-form solution* for zero-coupon bonds.

6 Operational semantics

So far we have only discussed *valuing* a contract. Another operation ripe for automated assistance is *executing*, or *managing*, a contract. Banks hold thousands of contracts, and it is no simple matter to remember when to pay out money, when money is due in, and when choices (for options) could be made.

Contracts *evolve* over time. For example, a contract that started life as

$$when\ (at\ t)\ (c_1\ `or`\ c_2)$$

evolves at time t into $(c_1\ `or`\ c_2)$, and hence either into c_1 or into c_2. This evolution is reminiscent of the way in which a program evolves over its execution run. Indeed, we can regard executing a contract as rather like running a program. Programming language folk use *operational semantics* to describe the evolution of a running program in a precise way, and we can do the same for contracts.

Space precludes a full treatment here. Suffice it to say that it is relatively easy to give rules that describe how a contract evolves over time, and implementing those rules in an automatic system is straightforward. The very same contract description that drives the contract valuation engine can also drive the contract management engine. As in the case of the valuation engine, the management engine must be enhanced only when new primitive combinators are added — it can deal automatically with arbitrary contracts built from the primitives it knows about.

Not only that, but the state of a partly-executed contract is still a contract term, and as such it can be fed into the contract valuation engine. Thus the bank can at any time value its entire "book" of partly-executed contracts simply by feeding them into its valuation engine.

7 Putting our work in context

At first sight, financial contracts and functional programming do not have much to do with each other. It has been a surprise and delight to discover that many of the insights useful in the design, semantics, and implementation of programming languages can be applied directly to the description, valuation, and management of contracts. The original idea was to apply functional programming to a realistic problem, and to compare our resulting program with the existing imperative version — but we have ended up with a radical re-thinking of how to describe and evaluate contracts.

Though there is a great deal of work on domain-specific programming languages (see [Hudak, 1996, van Deursen et al., 2000] for surveys), our work is virtually the only attempt to give a formal description to financial contracts. An exception is the RISLA language developed at CWI [van Deursen and Klint, 1998], an object-oriented domain-specific language for financial contracts. RISLA is designed for an object-oriented framework, and appears to be more stateful and less declarative than our system.

We have presented our design as a combinator library embedded in Haskell, and indeed Haskell has proved an excellent host language for prototyping both

the library design and various implementation choices. However, our design is absolutely not Haskell-specific. The big payoff comes from a declarative approach to *describing* contracts. As it happens we also used a functional language for *implementing* the contract language, but that is somewhat incidental. It could equally well be implemented as a free-standing domain-specific language, using domain-specific compiler technology.

Many application areas use one or more *ad hoc* languages to specify domain objects, although such languages are usually thought of as "data file formats" rather than a "language". The work we describe here is a good example of the power of the linguistic approach: not only can we describe a much richer family of contracts than any commercial system, but the language structure leads directly to a modular and robust software framework. Principled thinking has a practical payoff.

Acknowledgements

We warmly thank John Wisbey, Jurgen Gaiser-Porter, and Malcolm Pymm at Lombard Risk Systems Ltd for their collaboration. They invested a great deal of time in educating one of the present authors (Peyton Jones) in the mysteries of financial contracts and the Black-Derman-Toy valuation model. Jean-Marc Eber warmly thanks Philippe Artzner for many helpful discussions and Pierre Weis for numerous advices. We also thank Conal Elliott, Jeremy Gibbons, Andrew Kennedy, Stephen Jarvis, Andy Moran, Chris Okasaki, Norman Ramsey, Colin Runciman, Ganesh Sittampalam, David Vincent and the ICFP referees, for their helpful feedback.

References

[Boyle et al., 1997] Boyle, P., Broadie, M., and Glasserman, P. (1997). Monte carlo methods for security pricing. *Journal of Economic Dynamics and Control*, 21:1267–1321.

[Claessen and Sands, 1999] Claessen, K. and Sands, D. (1999). Observable sharing for functional circuit description. In Thiagarajan, P. and Yap, R., editors, *Advances in Computing Science (ASIAN'99); 5th Asian Computing Science Conference*, Lecture Notes in Computer Science, pages 62–73. Springer Verlag.

[Cook and Launchbury, 1997] Cook, B. and Launchbury, J. (1997). Disposable memo functions. In Launchbury, J., editor, *Haskell workshop*, Amsterdam.

[Cook et al., 1998] Cook, B., Launchbury, J., and Matthews, J. (1998). Specifying superscalar microprocessors in Hawk. In *Formal techniques for hardware and hardware-like systems*, Marstrand, Sweden.

[Cox et al., 1979] Cox, J. C., Ross, S. A., and Rubinstein, M. (1979). Option pricing: a simplified approach. *Journal of Financial Economics*, 7:229–263.

[Elliott and Hudak, 1997] Elliott, C. and Hudak, P. (1997). Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 263–273. ACM, Amsterdam.

[Finne et al., 1999] Finne, S., Leijen, D., Meijer, E., and Peyton Jones, S. (1999). Calling Hell from Heaven and Heaven from Hell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 114–125, Paris. ACM.

[Ho and Lee, 1986] Ho, T. and Lee, S. (1986). Term Structure Movements and Pricing Interest Rate Contingent Claims. *Journal of Finance*, 41:1011–1028.

[Hudak, 1996] Hudak, P. (1996). Building domain-specific embedded languages. *ACM Computing Surveys*, 28.

[Hughes, 1985] Hughes, J. (1985). Lazy memo-functions. In *Proc Aspenas workshop on implementation of functional languages.*

[Leroy et al., 1999] Leroy, X., Vouillon, J., Doligez, D., et al. (1999). The Objective Caml system, release 3.00. Technical Report, INRIA, available at http://caml.inria.fr/ocaml.

[Marlow et al., 1999] Marlow, S., Peyton Jones, S., and Elliott, C. (1999). Stretching the storage manager: weak pointers and stable names in Haskell. In Koopman, P. and Clack, C., editors, *International Workshop on Implementing Functional Languages (IFL'99)*, number 1868 in Lecture Notes in Computer Science, Lochem, The Netherlands. Springer Verlag.

[Musiela and Rutkowski, 1997] Musiela, M. and Rutkowski, M. (1997). *Martingale Methods in Financial Modelling.* Springer.

[Peyton Jones et al., 1999] Peyton Jones, S., Hughes, R., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C., and Wadler, P. (1999). Report on the programming language Haskell 98. `http:/haskell.org`.

[Revuz and Yor, 1991] Revuz, D. and Yor, M. (1991). *Continuous Martingales and Brownian Motion.* Springer.

[van Deursen et al., 2000] van Deursen, A., Kline, P., and Visser, J. (2000). Domain-specific languages: an annotated bibliography. Technical report, Centrum voor Wiskunde en Informatica, Amsterdam.

[van Deursen and Klint, 1998] van Deursen, A. and Klint, P. (1998). Little languages: little maintenance? *Journal of Software Maintenance*, 10:75–92.

[Willmot et al., 1993] Willmot, P., Dewyne, J., and Howison, S. (1993). *Option Pricing: Mathematical Models and Computation.* Oxford Financial Press.